The FoCus User Manual

Customizable format strings and directives, Version 1.0 beta 1 "Kokyu Ho"

Didier Verna `<didier@didierverna.net>`

Table of Contents

Copying

1 Introduction

`Format` is a powerful utility in the Common Lisp standard. Format strings are written in what can be considered as a printing DSL (Domain Specific Language). However powerful that language is though, it suffers from two important limitations.

1. It is not modifiable: only a predefined set of standard directive characters can be used and it is not possible to alter their semantics.
2. It is hardly extensible. The only embryonic extension mechanism available, the `~/` directive, is extremely cumbersome to use. The called function must either reside in the `cl-user` package, or you must always use its fully qualified name in the format string, even if the corresponding code lies in the appropriate package. For instance, consider that there is a function called `my-format-function` in the package named `:my.long.package.name`. Every time you want to use this function, you need to write something like

   ```
   (format t "~/my.long.package.name:my-format-function/" ...)
   ```

   which essentially makes the `/` directive unusable.

`FoCus` is a library designed to circumvent those limitations. It allows you to customize the `format` DSL by adding new directive characters or modifying the standard ones. The semantics of these directive characters is specified in a so-called *format table*, a concept very close to that of readtables. `FoCus` ultimately translates into regular `format` calls.

This is the `FoCus` user manual. `FoCus` also comes with a *Reference Manual*.

Chapter 3 [Quick Start], page 7 provides a very short example in order to give an overview of what's coming next. Chapter 4 [Usage], page 11 explains in detail how to use the library.

2 Installation

See Section A.3 [Supported Platforms], page 20, for information on portability or requirements. See `FoCus`'s homepage for tarballs, Git repository and online documentation. `FoCus` is also available via Quicklisp.

In order to install and load the bare Lisp library, unpack it somewhere in the ASDF 3 source registry and type this at the REPL:

```
(asdf:load-system :net.didierverna.focus)
```

Alternatively, if you just want to use the core library without the extensions (see Appendix A [Technical Notes], page 19), you can also just load the '`net.didierverna.focus.core`' system.

In addition to the bare Lisp library, the `FoCus` distribution offers documentation in the form of 2 different manuals. If you want to benefit from all those wonders, some bits of manual installation are needed. After unpacking somewhere in the ASDF 3 source registry, please perform the following steps, in order.

1. Edit `make/config.make` to your specific needs.
2. Type `make` to compile the documentation (user manual and lpossibly reference manual). By default, the documentation is built in info, PDF and HTML formats. If you want other formats (DVI and PostScript are available), type `make all-formats`. You can also type individually `make dvi` and/or `make ps` in order to get the corresponding format.
3. As documented in `make/config.make`, the reference manual is only generated if you have SBCL and the Declt library at hand (see `http://www.lrde.epita.fr/~didier/software/lisp/misc.php#declt`).
4. Type `make install` to install the documentation. If you have compiled the documentation in DVI and PostScript format, those will be installed as well.

Type `make uninstall` to uninstall the library.

3 Quick Start

In this chapter, we assume that you have properly installed `FoCus` (see Chapter 2 [Installation], page 5), and we build a short example to get you started.

Suppose you're writing a "quotation" library which outputs a lot of quotations, enclosed in (back)quotes like this: ‘`it's raining cats and dogs`’. If you're doing this very often, a pattern will quickly arise: `(format t "‘~A’" quotation)`, which you could abstract away with a custom formatting function called `quotation-formatter` for instance.

The problem, of course, is that writing this

```
(format t "~/quotation:quotation-formatter/" quotation)
```

is longer than doing it the quick'n dirty way, which is quite frustrating. That's where `FoCus` comes to the rescue. What if you had a new `format` directive, say, `~‘`, that would do the quotation of its argument? This is what we are going to do now.

3.1 Full Source

The complete source code is given below. You can just cut'n paste it in a REPL and it will (should) work. It is also contained in a file called `demos/quickstart.lisp` in the distribution (along with other demo programs). You can try it out with, *e.g.*, '`sbcl --script`' in a terminal.

```
(in-package :cl-user)

(require "asdf")
(asdf:load-system :net.didierverna.focus)
(net.didierverna.focus:nickname-package)


(defpackage :quotation
  (:use :cl)
  (:export :quotation))

(in-package :quotation)

(defun quotation-formatter (stream argument colonp atsignp &rest arguments)
  (declare (ignore colonp atsignp arguments))
  (write-char #\‘ stream)
  (write-string argument stream)
  (write-char #\’ stream))

(let ((table (focus:make-format-table)))
  (focus:with-format-table table
    (focus:set-format-directive #\‘ :function ’quotation-formatter))

  (defun quotation (who quotation)
    (focus:with-format-table table
      (focus:format t "As ~A would say: ~‘.~%" who quotation))))


(in-package :cl-user)

(quotation:quotation "Bugs Bunny" "Errr, what’s up Doc?")
```

3.2 Explanation

Let's examine this program step-by-step now.

First, we put ourselves in the Common Lisp user package, and load `FoCus` from its ASDF system '`net.didierverna.focus`'. `FoCus` lives in a package also named '`net.didierverna.focus`', that we nickname to just `focus` immediately, thanks to the function `nickname-package`.

```
(in-package :cl-user)

(require "asdf")
(asdf:load-system :net.didierverna.focus)
(net.didierverna.focus:nickname-package)
```

Next, we create our own `:quotation` package (don't do that for real!), and provide a `quotation-formatter` function that will write its string argument between a pair of (back)quotes. Note that this is the kind of function that you would use in the standard `~/` format directive.

```
(defpackage :quotation
  (:use :cl)
  (:export :quotation))

(in-package :quotation)

(defun quotation-formatter (stream argument colonp atsignp &rest arguments)
  (declare (ignore colonp atsignp arguments))
  (write-char #\‘ stream)
  (write-string argument stream)
  (write-char #\’ stream))
```

The interesting part comes now. `FoCus` uses so-called *format tables* to store the mappings between directive characters and their behavior. Much like what a readtable does with macro characters. So let's create a new format table. This is done with the function `make-format-table`.

```
(let ((table (focus:make-format-table)))
```

By default, new format tables inherit the standard `format` behavior (that is, all standard directives are recognized). Let's add a new directive character, ‘ (backquote) to our new format table, and map it to the `quotation-formatter` function. This is done with the function `set-format-directive`. By default, `FoCus` works with the so-called *current format table*. One way of making sure that the current format table is the appropriate one is to use the macro `with-format-table`.

```
  (focus:with-format-table table
    (focus:set-format-directive #\‘ :function ’quotation-formatter))
```

Now, let's create a silly `quotation` function for printing someone's quote. `FoCus` provides its own `format` function which wraps around the standard one. This function works exactly like the original `format`, except that it uses the current format table to interpret your custom format string, so again, we need to make sure that the proper table is used when the function is called.

```
  (defun quotation (who quotation)
    (focus:with-format-table table
      (focus:format t "As ~A would say: ~‘.~%" who quotation))))
```

Finally, let's try it!

```
(in-package :cl-user)

(quotation:quotation "Bugs Bunny" "What’s Up Doc?")
```

4 Usage

Just when you thought format strings were unreadable, it's going to get a whole lot crazier. With `FoCus`, you can not only create your own cryptic format directives, but you can also completely mess up the standard ones, which is where the fun really begins.

First of all, `FoCus` itself resides in a package called `net.didierverna.focus`. You can automatically nickname this package with the following function.

`nickname-package` **&optional** *NICKNAME* [Function]
Add *NICKNAME* (`:focus` by default) to the `:net.didierverna.focus` package.

Using `FoCus` involves three steps: creating format tables, filling them with format directives and actually using them in calls to `format` or `formatter`.

4.1 Creating Format Tables

In order to use custom format strings, you first need to create a so-called *format table*. A format table stores mappings between format directives (represented by their directive characters) and their intended meaning. In essence, a format table is very similar to a readtable: it describes how `FoCus` is to interpret your custom format strings in order to translate them into standard ones. Creating a format table is done by calling the function `make-format-table`.

`make-format-table` **&optional** (*INITIALLY* **:standard**) [Function]
Create and return a new format table.

The optional argument *INITIALLY* lets you decide what you want the table to look like when it's created. A value of `:standard` creates a format table behaving exactly like standard `format`. `:standard-downcase` and `:standard-upcase` do the same, except that when applicable, only the downcase (respectively upcase) directive characters are added (See also Section 4.3.2 [Casing], page 13). Finally, a `:blank` format table does not contain any directive at all.

4.2 Interlude: Referring to Format Tables

Gosh, that was a lot to digest, I know. Sorry. Let's pause for a moment and catch our breath.

Whether it is to modify them or to actually use them, the rest of the library needs to refer to format tables in one way or another. You refer to a specific format table by using a so-called *format table designator*. Format table designators can be of various forms.

First of all, it is possible to use a table object directly (as returned by `make-format-table`). You may also refer to a format table by name, if you have *registered* it (see Section 4.2.1 [Table Registration], page 11) and finally, if you don't specify anything, the so-called *current format table* is used (see Section 4.2.2 [Current Format Table], page 12).

4.2.1 Table Registration

If you plan to manipulate several format tables in the same application, you may want to give them names. This process is called *registration*. You can register a format table with the `register-format-table` function. The opposite (un-naming a table, if you prefer) is done with the `unregister-format-table` function.

`register-format-table` *TABLE NAME* **&optional** *FORCE* [Function]
Register *TABLE* under *NAME* (a symbol) and return it.

If a table is already registered under that name, `FoCus` throws a `table-collision` error, unless the optional *FORCE* argument is non-`nil`.

`unregister-format-table` *NAME* [Function]
Unregister *NAME*d table.

4.2.2 Current Format Table

Unless otherwise specified, `FoCus` uses a so-called *current format table* every time it needs one. This table is stored in the global variable `*format-table*`, which you are free to use directly.

`FoCus` also provides a couple of macros to manipulate the current format table in a very idiomatic way.

`with-format-table` *TABLE-OR-NAME* **&body** *BODY* [Macro]
Execute *BODY* with the current format table bound to *TABLE-OR-NAME*.

TABLE-OR-NAME is of course a table designator: either a table object directly, or the name of a previously registered table.

When the `flv` extension is available (see Section A.2 [Optional Features], page 19) `FoCus` also defines a macro called `in-format-table`. You may check for the availability of this macro by testing the presence of the feature `:net.didierverna.focus.flv`.

`in-format-table` *TABLE-OR-NAME* [Macro]
Set the current format table to *TABLE-OR-NAME* in the current file.

TABLE-OR-NAME is again a table designator: either a table object directly, or the name of a previously registered table.

This macro is meant to be used in a way similar to `in-package` or `in-readtable`. It actually works the same way and lets you write highliy idiomatic code like this:

```
(in-package      :my.library.name)
(in-format-table :my.library.name)
```

Remember however that there is a big difference between these two calls. In the case of `in-package`, the keyword is a *string designator*. Your package really is named `"MY.LIBRARY.NAME"`. In the case of `in-format-table`, your table really is named after the symbol `MY.LIBRARY.NAME` interned into the `KEYWORD` package.

Finally, note that using `in-format-table` requires that the format table in question be available at compile-time (see Section 4.5 [Compile or Run Time], page 14).

4.3 Modifying Format Tables

Customizing the contents of your format tables involves a single function: `set-format-directive`.

`set-format-directive` *CHAR* **&key** *STANDARD FUNCTION* [Function]
(*BOTH-CASE* **t**) *FORCE* (*TABLE* ***format-table***)
Set a ~*CHAR* directive in *TABLE*.

TABLE is the format table to modify. It defaults to the current format table, and may otherwise be specified by passing a table designator as value to the `:table` keyword argument.

4.3.1 Directive Characters

CHAR is the character for which you want to create a new directive (Chapter 3 [Quick Start], page 7 used `#\‘` as an example). Note that `FoCus` allows you to modify standard directives as well as creating new ones. This means that you can create a completely new directive, freely override a standard directive with a new meaning, alias a standard directive to a new character, anything you want.

If such a directive is already set in *TABLE*, `FoCus` throws a `table-directive-collision` error, unless you explicitely pass a non-`nil` value to the `:force` keyword argument.

4.3.2 Casing

In the Common Lisp standard, the case of a directive character is ignored. On the contrary, case *does* matter to `FoCus`. A format table contains separate entries for upcase and downcase characters (when applicable of course). By the way, this begins as soon as you create a new table (see Section 4.1 [Creating Format Tables], page 11 and remember the optional *INITIALLY* argument to `make-format-table`).

By default, the behavior of `set-format-directive` conforms to that of the standard however: when you set a new directive character that has both an upcase and a downcase version, both versions get the definition.

You can change this behavior by passing a `nil` value to the `:both-case` keyword argument, hence distinguishing between case versions. As an example, consider the case[1] where you find yourself short of directive characters. What you can do is retain the standard meaning for all the upcase versions and define new custom directives for the downcase ones. Or the other way around. yOu gEt tHe iDeA.

4.3.3 Directive Definition

There are currently two ways to define new directives in `FoCus`: you can either assign them the meaning of a standard directive, or associate them with a custom formatter.

4.3.3.1 Standard Directives

In order to use a standard directive, use the `:standard` keyword with the corresponding character as value. For instance, calling '`(set-format-directive #\Y :standard #\S)`' will result in making `~Y` (and `~y` by default, see Section 4.3.2 [Casing], page 13) have the same meaning as the standard `~S` directive. By the way, this is how a newly created format table is made `:initially :standard`: for every standard directive character `#\c`, a call to '`(set-format-directive #\c :standard #\c)`' is issued.

One thing about aliasing standard directives is worth mentioning here. `FoCus` is aware of the syntactic specificities of the standard directives. More precisely, we can split standard directives in three categories: *simple* directives like `~S`, *grouping* directives like `~{~}` which use a couple of characters and *delimiting* directives in which there is only one guy: the `~/` directive.

`FoCus` does nothing special about simple and grouping directives. In particular, you can very well alias only one of the two characters of a standard grouping directive (see, it's pretty easy to completely mess up the standard set of directives). Or, to put it differently, creating a new grouping directive couple would involve *two* calls to `set-format-directive`, for example like this:

```
(set-format-directive #\‘ :standard #\{)
(set-format-directive #\’ :standard #\})
```

On the other hand, `FoCus` understands when you alias the `~/` directive, and expects your custom format strings to be syntactically correct. For instance, if you alias `~!` to `~/`, then your format strings should look like this: `"~!foo:bar!"`. It is *not* possible to match a `#\/` with a `#\!`, in any order.

[1] so to speak. . .

4.3.3.2 Custom Formatters

In order to associate a directive character with a custom formatter function, use the `:function` keyword instead of the `:standard` one (see Chapter 3 [Quick Start], page 7 for an example). It is expected that you provide a *function designator* as a value, that is, a function name, object, or even a lambda expression.

The function in question must comply with the regular formatter protocol, as such custom directives are eventually translated into standard `~/` ones.

4.3.4 Directive Removal

In order to remove a directive from a format table, simply call `set-format-directive` without any definition, that is, without using the `:standard` or `:function` keyword arguments. For instance, a call to '`(set-format-directive #\s)`' will remove the definition for both `~s` and `~S` (but then again, see Section 4.3.2 [Casing], page 13) from the current format table.

4.4 Using Format Tables

`FoCus` provides its own version of `format`.

`format` *DESTINATION FORMAT-CONTROL* **&rest** *ARGS* [Macro]
Wrapper around the standard `format` function.

`FoCus`'s `format` wraps around the standard one. When *FORMAT-CONTROL* is a format string, it is translated into a standard one according to the current format table. That's it.

The reason why `FoCus`'s `format` is a macro will be explained in Section 4.5 [Compile or Run Time], page 14.

4.5 Compile or Run Time

Normally, `FoCus` behaves dynamically, meaning that the translation to standard `format` calls occurs at run-time. This is the default behavior because it allows for maximum flexibility. . . and weirdness, admitedly, as the same call to `FoCus`'s `format` may behave differently if the current format table has changed in the meantime. The drawback of this approach, however, is that it induces a run-time overhead, which may become undesirable in format-intensive applications.

One way around this is to have your format table known at compile-time and instruct `FoCus` to perform a compile-time translation instead. This way, all dynamic trace of `FoCus` will disappear from your application. This can be done by setting the `*compile*` flag to a non-nil value (at compile-time of course!). By the way, this also explains why `format` is a macro instead of a regular function in `FoCus`.

When the `flv` extension is available (see Section A.2 [Optional Features], page 19), this variable is automatically made file-local, and `in-format-table` sets it to `t`, so that you don't have anything to do to switch to compile-time behavior. The `demos/quotation` library in the distribution provides an example of using `FoCus` in such a way.

Finally, when compile-time behavior is switched on, you may also use `FoCus`'s wrapper around the `formatter` macro.

`formatter` *FORMAT-STRING* [Macro]
Wrapper around the standard FORMATTER macro.

5 Miscellaneous

This section contains information about different features that are present in `FoCus` because of design decisions, but that I expect to be used only rarely, if at all.

5.1 Version Numbering

As `FoCus` evolves over time, you might one day feel the need for conditionalizing your code on the version of the library.

The first thing you can do to access the current version number of `FoCus` is use the `version` function.

`version` **&optional** (*TYPE* **:number**) [Function]

Return the current version number of `FoCus`. *TYPE* can be one of `:number`, `:short` or `:long`. For `:number`, the returned value is a fixnum. Otherwise, it is a string.

A `FoCus` version is characterized by 4 elements as described below.

- A major version number stored in the parameter `*release-major-level*`.
- A minor version number, stored in the parameter `*release-minor-level*`.
- A release status stored in the parameter `*release-status*`. The status of a release can be `:alpha`, `:beta`, `:rc` (standing for “release candidate”) or `:patchlevel`. These are in effect 4 levels of expected stability.
- A status-specific version number stored in the parameter `*release-status-level*`. Status levels start at 1 (alpha 1, beta 1 and release candidate 1) except for stable versions, in which case patch levels start at 0 (*e.g.* 2.4.0).

In addition to that, each version of `FoCus` (in the sense *major.minor*, regardless of the status) has a name, stored in the parameter `*release-name*`. The general naming theme for `FoCus` is “Aïkido movements”.

Here is how the `version` function computes its value.

- A version `:number` is computed as *major* . *10000* + *minor* . *100* + *patchlevel*, effectively leaving two digits for each level. Note that alpha, beta and release candidate status are ignored in version numbers (this is as if the corresponding status level was considered to be always 0). Only stable releases have their level taken into account.
- A `:short` version will appear like this for unstable releases: 1.3a4, 2.5b8 or 4.2rc1. Remember that alpha, beta or release candidate levels start at 1. Patchlevels for stable releases start at 0 but 0 is ignored in the output. So for instance, version 4.3.2 will appear as-is, while version 1.3.0 will appear as just 1.3.
- A `:long` version is expanded from the short one, and includes the release name. For instance, 1.3 alpha 4 "Kote Gaeshi", 2.5 beta 8 "Irimi Nage", 4.2 release candidate 1 "San Kyo" or 4.3.2 "Suwari Wasa Shomen Uchi Ikkyo". As for the short version, a patchlevel of 0 is ignored in the output: 1.3 "Ju Wasa".

6 Conclusion

So that's it I guess. You know all about `FoCus` now. The next step is to actually use it and make your format strings even less readable than the original ones.

Now, go my friend. Go obfuscate your Lisp printing code!

Appendix A Technical Notes

This chapter contains important information about the library's configuration and optional features.

A.1 Configuration

Some aspects of `FoCus`'s behavior can be configured *before* the library system is actually loaded. `FoCus` stores its user-level configuration (along with some other setup parameters) in another ASDF system called '`net.didierverna.focus.setup`' (and the eponym package). In order to configure the library (I repeat, prior to loading it), you will typically do something like this:

```
(require "asdf")
(asdf:load-system :net.didierverna.focus.setup)
(net.didierverna.focus.setup:configure <option> <value>)
```

`configure` *KEY VALUE* [Function]

Set *KEY* to *VALUE* in the current `FoCus` configuration.

Out of curiosity, you can also inquire the current configuration for specific options with the following function.

`configuration` *KEY* [Function]

Return *KEY*'s value in the current `FoCus` configuration.

Currently, the following options are provided.

`:swank-eval-in-emacs`

This option is only useful if you use Slime, and mostly if you plan on hacking `FoCus` itself. The library provides indentation information for some of its functions directly embedded in the code. This information can be automatically transmitted to (X)Emacs when the ASDF system is loaded if you set this option to `t`. However, note that for this to work, the Slime variable `slime-enable-evaluate-in-emacs` must also be set to `t` in your (X)Emacs session. If you're interested to know how this process works, I have described it in the following blog entry: `http://www.didierverna.net/blog/index.php?post/2011/07/20/One-more-indentation-hack`.

`:restricted`

Some features of `FoCus` require external functionality that may not be available in all contexts. Normally, `FoCus` should autodetect this and switch to so-called *restricted mode* at build-time (see Section A.2 [Optional Features], page 19). If `FoCus` has failed to autodetect the problem (in which case I would like to know), or if for some reason, you explicitly want to disable those features, you may set the `:restricted` configuration option to `t`. Another way to do it, without even bothering with configuration is to just use the '`net.didierverna.focus.core`' system instead of the regular one.

A.2 Optional Features

As seen in Section 4.2.2 [Current Format Table], page 12, the current format table is stored in the variable `*format-table*`. In many regards, this variable looks very much like `*package*` or `*readtable*` and we sure would like it to behave in the same way, notably with respect to `load` and `compile-file`.

Standard Common Lisp doesn't allow this, but fortunately, there is one workaround. ASDF has an extension called '`asdf-flv`' (`http://www.lrde.epita.fr/~didier/software/lisp/`

misc.php#asdf-flv), also written by me, which essentially allows one to define any number of *file-local* variables, behaving like *package* or *readtable* with respect to load and compile-file.

Normally, FoCus automatically detects the availability of this extension, and if applicable, makes the variable *format-table* file-local and defines the in-format-table macro. It also puts :net.didierverna.focus.flv on *features* so that you may conditionalize your code accordingly.

FoCus can be used without this extension, but if you want to make it a strong requirement, you may load the system 'net.didierverna.focus.flv' instead of the regular one. Loading it will fail if asdf-flv isn't available.

A.3 Supported Platforms

FoCus is an ASDF 3 library. It doesn't have any system, platform or compiler-specific requirement, so portability problems should be regarded as bugs, and reported to me (please). This includes potential problems using either the standard or modern version of Allegro Common Lisp.

FoCus optionally depends on 'asdf-flv' for providing the in-format-table macro (see Section A.2 [Optional Features], page 19).

Appendix B Indexes

B.1 Concepts

B.2 Functions

B.3 Variables

B.4 Data Types